

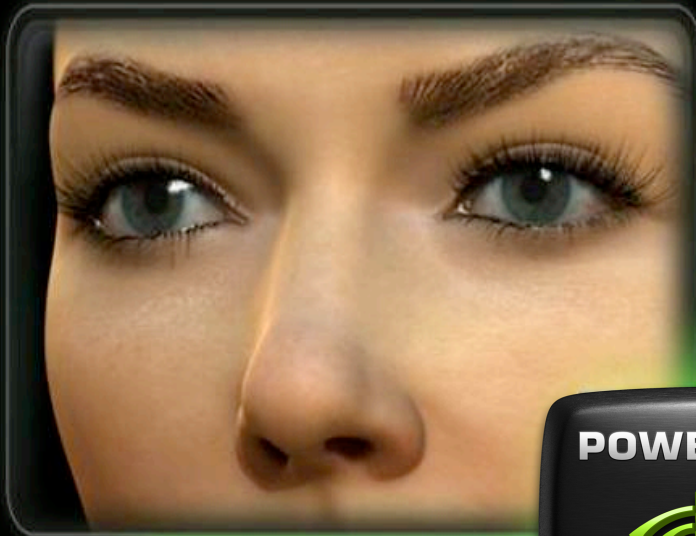
Heterogeneous Parallel Computing

CUDA C and OpenCL

Chris Lamb

nvidia.com/cuda

Graphics Processing Unit



Crysis © 2006 Crytek / Electronic Arts

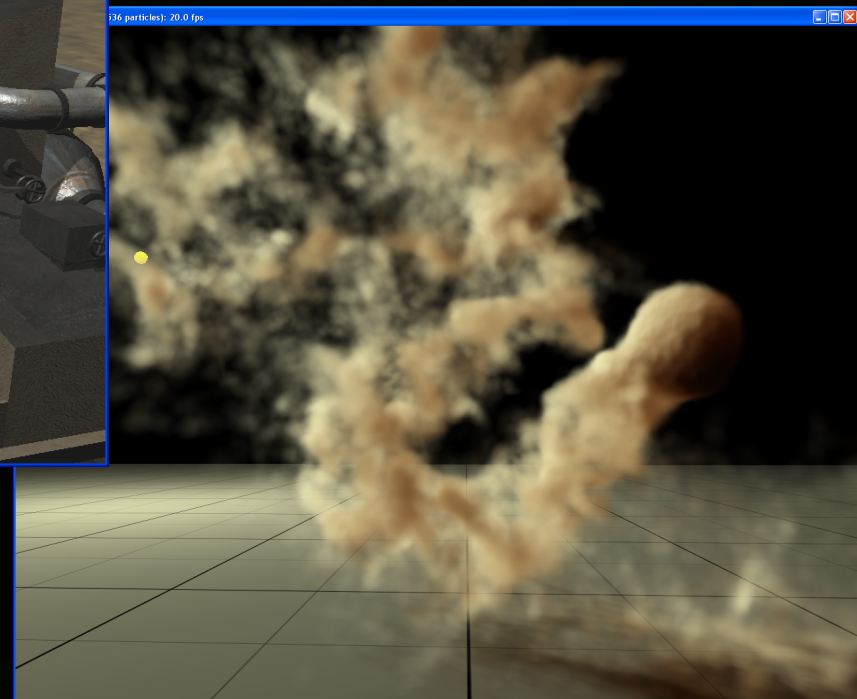


Hellgate: London © 2005-2006 Flagship Studios, Inc. Licensed by NAMCO BANDAI Games America, Inc.



Full Spectrum Warrior: Ten Hammers © 2006 Pandemic Studios, LLC. All rights reserved. © 2006 THQ Inc. All rights reserved.

“GPU Computing” makes you think of this...

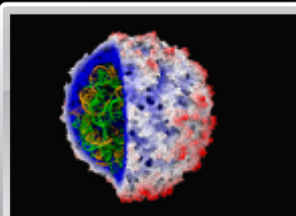


But with CUDA you also get...



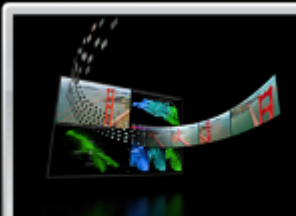
146X

Interactive visualization
of volumetric white
matter connectivity



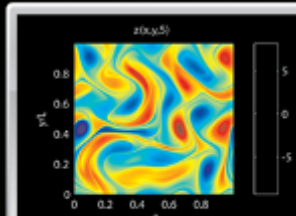
36X

Ionic placement for
molecular dynamics
simulation on GPU



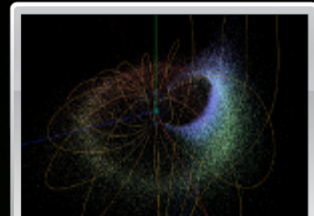
19X

Transcoding HD video
stream to H.264



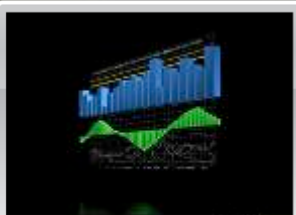
17X

Simulation in Matlab
using .mex file CUDA
function



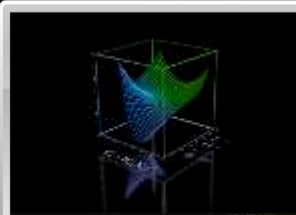
100X

Astrophysics N-body
simulation



149X

Financial simulation of
LIBOR model with
swaptions



47X

GLAME@lab: An M-
script API for linear
Algebra operations on
GPU



20X

Ultrasound medical
imaging for cancer
diagnostics



24X

Highly optimized object
oriented molecular
dynamics



30X

Cmatch exact string
matching to find similar
proteins and gene
sequences

GPUs Are Fast



1995
NV1
1M Trans



1999
GeForce 256
22M Trans



2002
GeForce 4
63M Trans



2003
GeForce FX
130M Trans



2004
GeForce 6
222M Trans



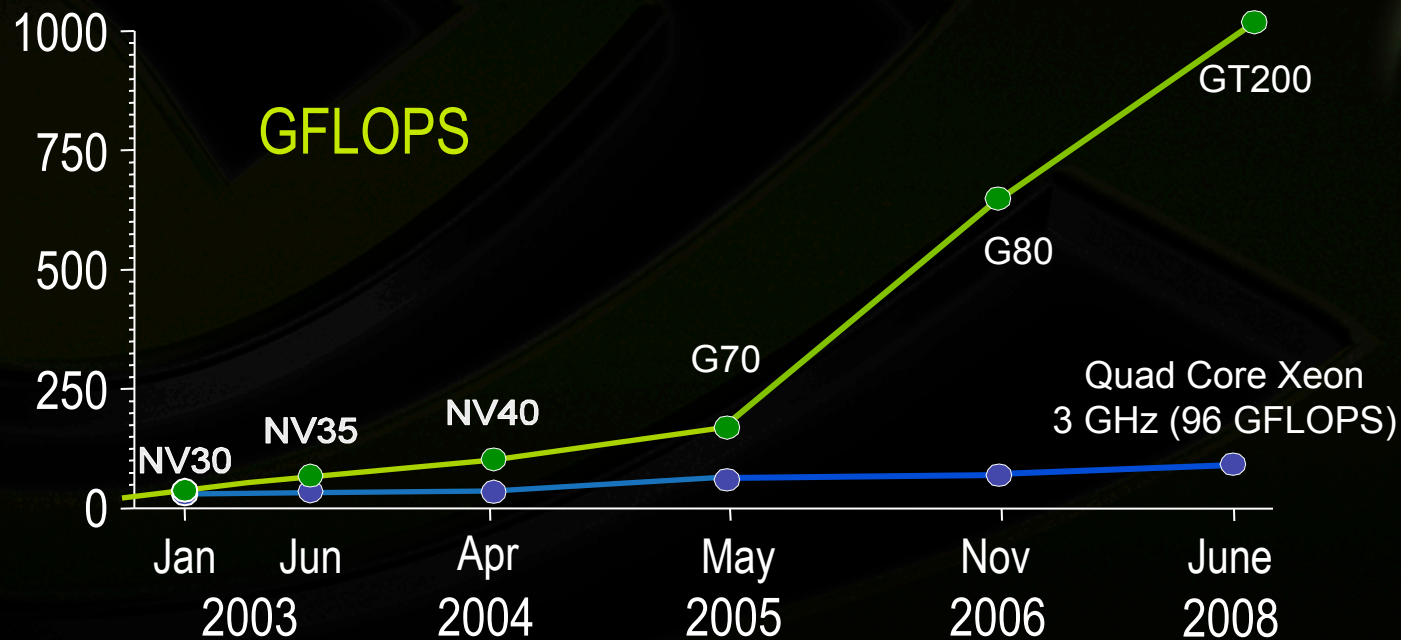
2005
GeForce 7
302M Trans



2006-2007
GeForce 8
754M Trans



2008
GeForce GTX 200
1.4 Billion Transistors



GT200 – Consumer Supercomputer



- **GT200 theoretical peak performance:**
 - 1 TFLOPS single precision
 - 87 GFLOPS double precision
 - Raw memory bandwidth: ~142 GB/sec
 - **Linpack (DGEMM) 95% of peak: sustained 82.4 GFLOPS**
 - **8 node (8U!) cluster achieves Linpack Rmax 1.25 TFLOPS**
- **Video game market subsidizes R&D**
 - > 100 million units shipped
 - Economies of scale at work: GTX295 costs <\$500!
 - All NVIDIA cards since 2006 support CUDA



Building a 100TF datacenter



CPU 1U Server



4 CPU cores

0.07 Teraflop

\$ 2000

400 W

1429 CPU servers

\$ 3.1 M

571 KW

4 GPUs: 960 cores

4 Teraflops

\$ 8000

700 W

25 CPU servers
25 Tesla systems

\$ 0.31 M

27 KW

Tesla 1U System



10x lower cost

21x lower power

CPU

GPU



“Fast” = High Throughput

- How to design an architecture for **throughput?**
- **Throughput vs Latency**
- **Maximize parallelism**
- **Graphics is a throughput problem**
- **So is scientific computing and many other important and emerging problems!**

The Trend

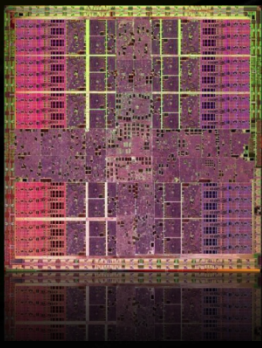
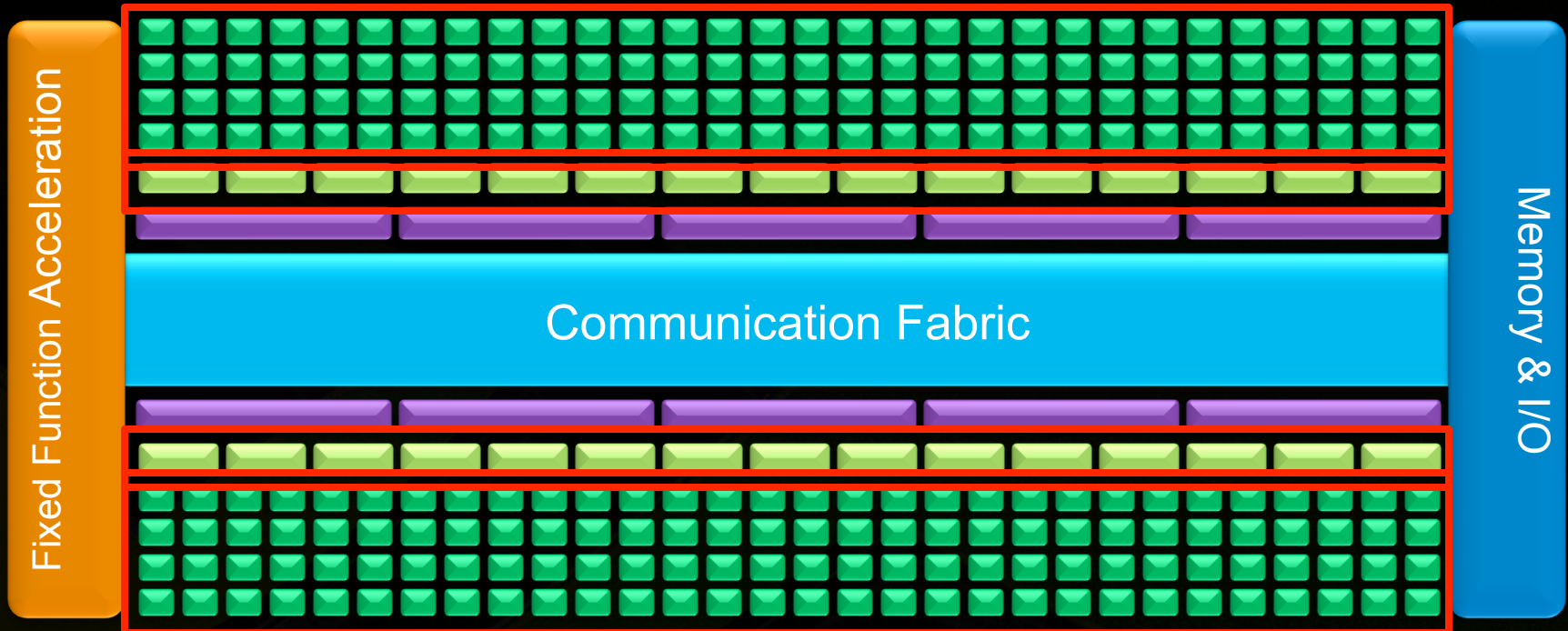


- Era of faster CPUs is over
 - There won't be 10GHz chips
- “Moore's Law” => more transistors => wider units
- It's got to be parallel to be fast
- **Not just NVIDIA party line – everyone's saying this**
- GPUs are already at where CPU are going
 - CPU today = 8 cores
 - GT200 = **240 cores**

NVIDIA GPU Architecture



GT200

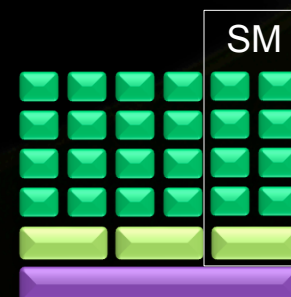


- 240 scalar cores
- On-chip memory
- Texture units



Streaming Multiprocessor (SM)

- **8 scalar cores (SP) per SM**
 - 16K 32-bit registers (64KB)
 - usual ops: float, int, branch, ...
 - special ops: exp, sin, cos, sqrt, mul24, saturate, ...
 - synchronization support
- **Shared double precision unit**
 - IEEE 754 64-bit floating point
 - fused multiply-add
 - full-speed denorm. operands and results
- **Direct load/store to memory**
 - high bandwidth (~142 GB/sec)
- **Low-latency on-chip memory**
 - 16KB available per SM
 - shared amongst threads of a block
 - supports thread communication



Key Architectural Ideas



- **SIMT** (Single Instruction Multiple Thread) **execution**
 - threads run in groups of 32 called **warps**
 - threads in a warp share instruction unit (IU)
 - 1 instruction x 32 threads issued in 4 clocks
 - HW automatically handles divergence
- **Hardware multithreading**
 - HW resource allocation & thread scheduling
 - **HW relies on threads to hide latency**
 - any warp not waiting for something can run
 - context switching is (basically) free



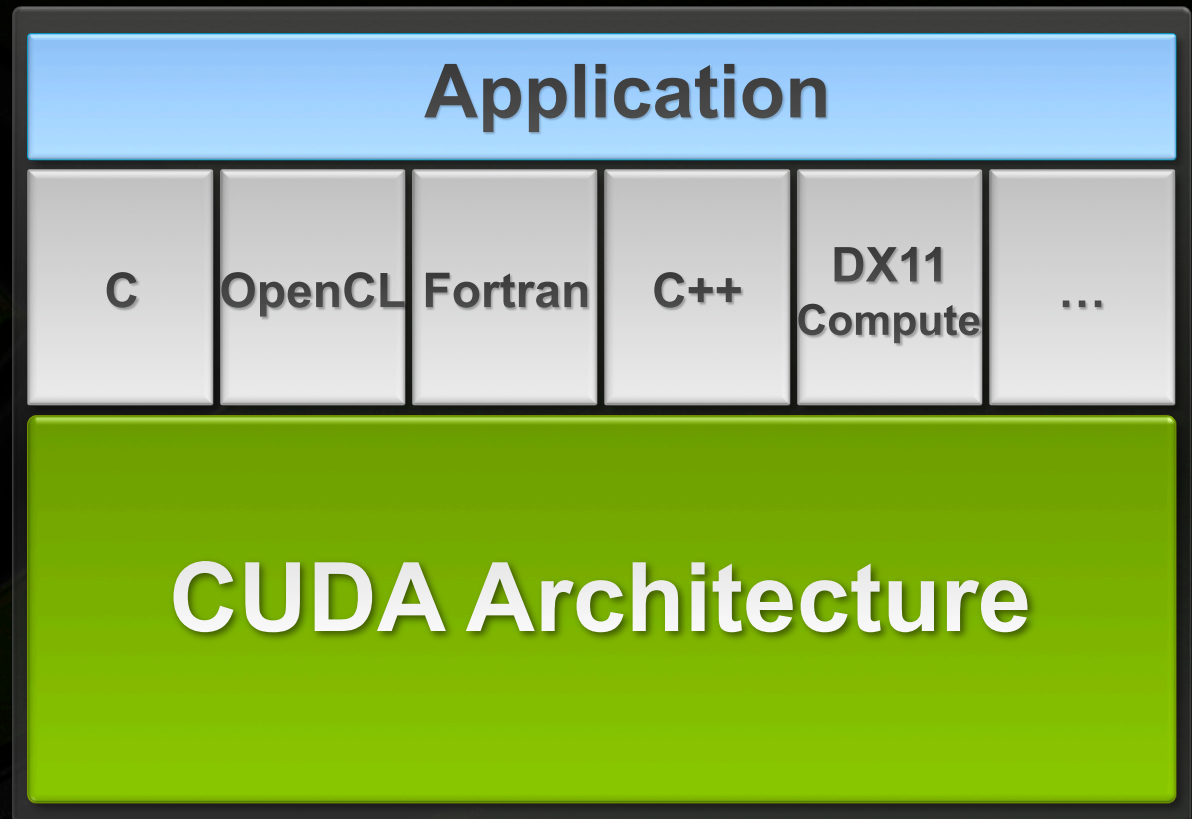
How do you write fast code?

- **Task parallelism is short lived...**
 - More cores, more memory latency
- **Data parallel is the future**
 - Express a problem as data parallel....
 - Maps automatically to a scalable architecture
- **CUDA architecture provides an insight into a data parallel future**

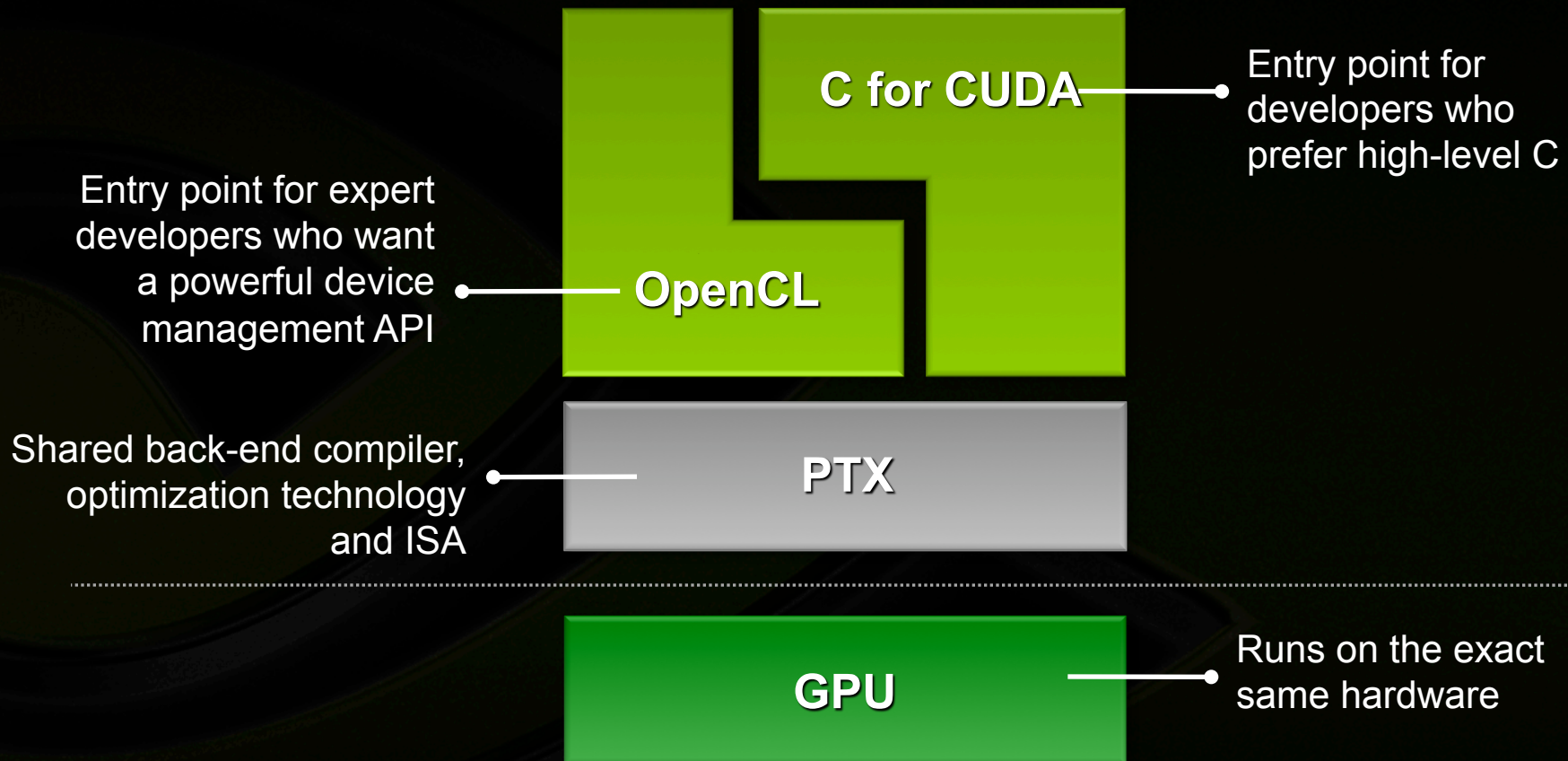
CUDA – NVIDIA's Parallel Computing Architecture



- PTX ISA and hardware compute engine
- Includes a C-compiler & support for OpenCL and DX11 Compute
- Architected to natively support all computational interfaces (standard languages and APIs)



OpenCL and C for CUDA



C for CUDA

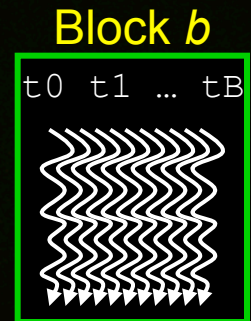


Hierarchy of concurrent threads



- Parallel **kernel** run by many threads
 - all threads execute the same piece of code
 - threads can take different paths through code
- Threads are grouped into **thread blocks**
 - threads in the same block can cooperate
- Threads/blocks have unique IDs

Thread t

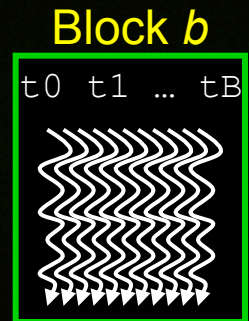
A single white wavy line with an arrow at the bottom, representing the execution path of a single thread.

Memory hierarchy



- Thread has **registers**
- Block has **shared memory**
 - Fast access by threads (fast like registers/cache)
- Device has off-chip **global memory**
 - Same DRAM used to store images for graphics
- memcpy over PCIe to/from **host memory**

Thread t

A white wavy arrow pointing downwards, representing a thread's access path.

Example: Vector Addition Kernel



Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>>(d_A, d_B, d_C);
}
```



Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>>(d_A, d_B, d_C);
}
```




Example: Host code for `vecAdd`

```
// allocate and initialize host (CPU) memory  
float *h_A = ..., *h_B = ...;
```

```
// allocate device (GPU) memory  
float *d_A, *d_B, *d_C;
```

```
cudaMalloc( (void**) &d_A, N * sizeof(float));  
cudaMalloc( (void**) &d_B, N * sizeof(float));  
cudaMalloc( (void**) &d_C, N * sizeof(float));
```

```
// copy host memory to device
```

```
cudaMemcpy( d_A, h_A, N * sizeof(float),  
            cudaMemcpyHostToDevice );  
cudaMemcpy( d_B, h_B, N * sizeof(float),  
            cudaMemcpyHostToDevice );
```

```
// execute the kernel on N/256 blocks of 256 threads each  
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Why is this different from a CPU?



- Different goals produce different designs
 - GPU assumes work load is highly parallel
 - CPU must be good at everything, parallel or not
- CPU: **minimize latency** experienced by 1 thread
 - big on-chip caches
 - sophisticated control logic
- GPU: **maximize throughput** of all threads
 - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
 - multithreading can hide latency => skip the big caches
 - amortize cost of control logic via SIMT

OpenCL



CUDA C and OpenCL Programming Styles



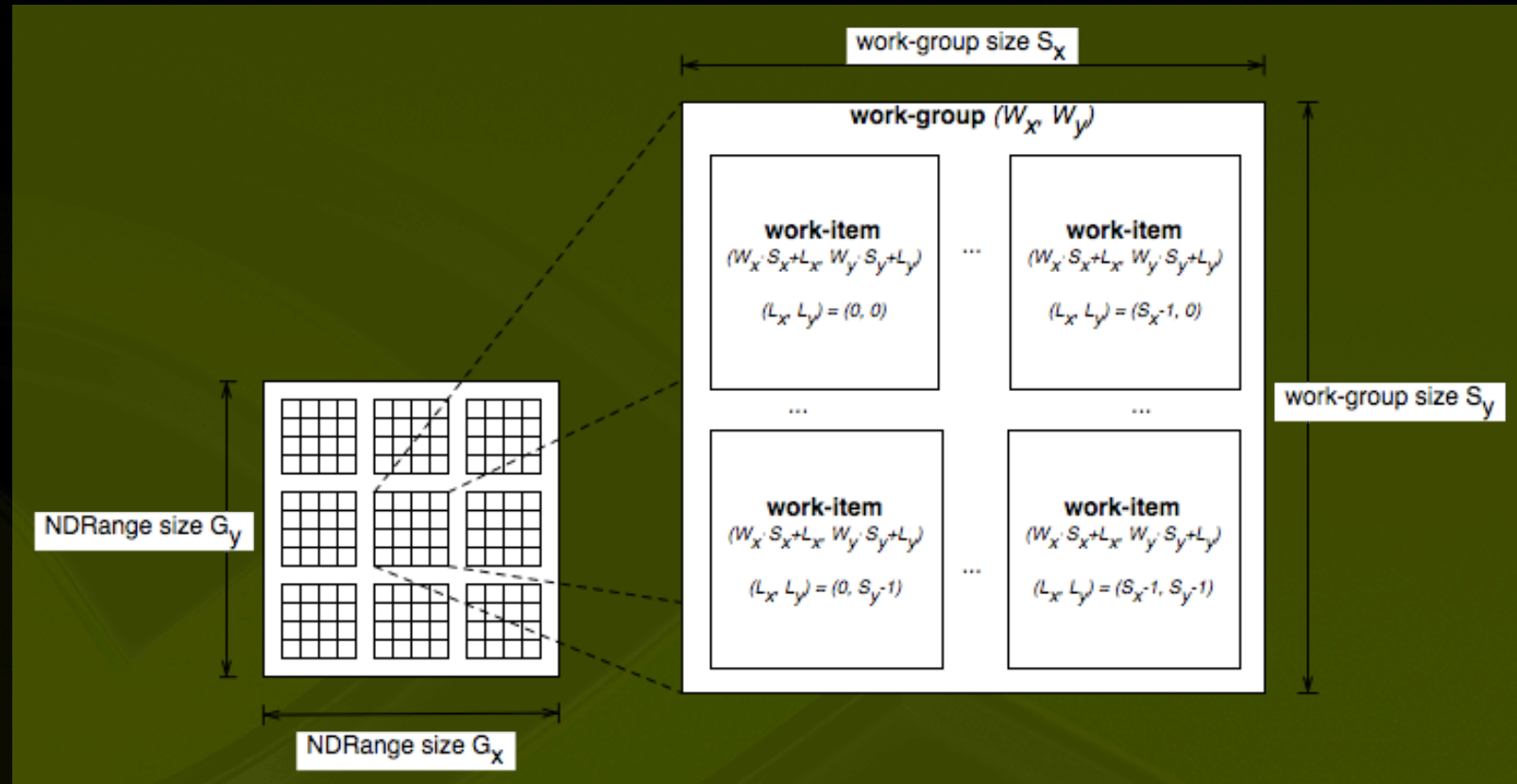
- **C for CUDA**

- C with parallel keywords
- C runtime that abstracts driver API
- Memory managed by C runtime
- Generates PTX

- **OpenCL**

- Hardware API - similar to OpenGL
- Programmer has complete access to hardware device
- Memory managed by programmer
- Generates PTX

Kernel Execution

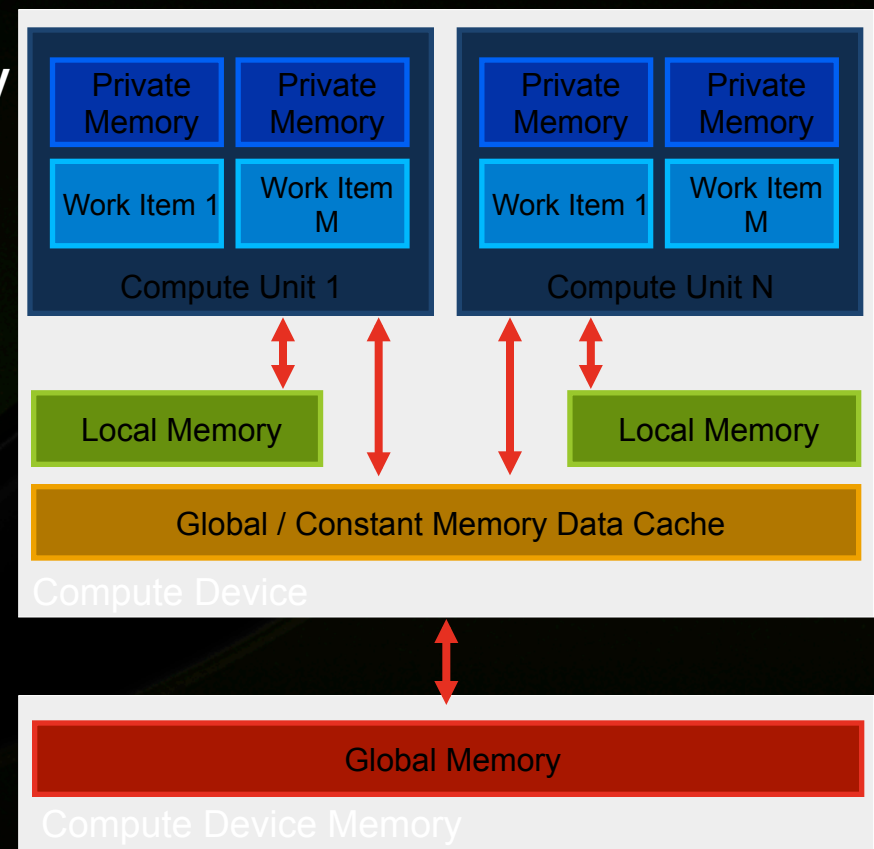


- Total number of work-items = $G_x \times G_y$
- Size of each work-group = $S_x \times S_y$
- Global ID can be computed from work-group ID and local ID

OpenCL Memory Model (Section 3.3)



- **Shared memory model**
 - Relaxed consistency
- **Multiple distinct address spaces**
 - Address spaces can be collapsed depending on the device's memory subsystem
- **Address spaces**
 - Private - private to a *work-item*
 - Local - local to a *work-group*
 - Global - accessible by all work-items in all work-groups
 - Constant - read only global space
- **Implementations map this hierarchy**
 - To available physical memories





VecAdd: Create Memory Objects

```
cl_mem memobjs[3];

// allocate input buffer memory objects
memobjs[0] = clCreateBuffer(context,
                           CL_MEM_READ_ONLY |    // flags
                           CL_MEM_COPY_HOST_PTR,
                           sizeof(cl_float)*n,    // size
                           srcA,                   // host pointer
                           NULL);                  // error code

memobjs[1] = clCreateBuffer(context,
                           CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           sizeof(cl_float)*n, srcB, NULL);

// allocate input buffer memory object
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                           sizeof(cl_float)*n, NULL, NULL);
```



VecAdd: Program and Kernel

```
// create the program
cl_program program = clCreateProgramWithSource(
    context,
    1,                      // string count
    &program_source,        // program strings
    NULL,                   // string lengths
    NULL);                  // error code

// build the program
cl_int err = clBuildProgram(program,
    0,                      // num devices in device list
    NULL,                   // device list
    NULL,                   // options
    NULL,                   // notifier callback function ptr
    NULL);                  // user data

// create the kernel
cl_kernel kernel = clCreateKernel(program, "vec_add", NULL);
```



VecAdd: Set Kernel Arguments

```
// set "a" vector argument
err = clSetKernelArg(kernel,
                      0,                                // argument index
                      (void *) &memobjs[0],             // argument data
                      sizeof(cl_mem));                  // argument data size

// set "b" vector argument
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1], sizeof(cl_mem));

// set "c" vector argument
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2], sizeof(cl_mem));
```

Spec

Setting kernel arguments:	Section 5.5.2
Executing Kernels:	Section 6.1
Reading, writing, and copying buffer objects:	Section 5.2.2

VecAdd: Invoke Kernel, Read Output



```
size_t global_work_size[1] = n; // set work-item dimensions

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel,
                              1,          // Work dimensions
                              NULL,       // must be NULL (work offset)
                              global_work_size,
                              NULL,       // automatic local work size
                              0,          // no events to wait on
                              NULL,       // event list
                              NULL);      // event for this kernel

// read output array
err = clEnqueueReadBuffer(context, memobjs[2],
                           CL_TRUE,      // blocking
                           0,             // offset
                           n*sizeof(cl_float), // size
                           dst,           // pointer
                           0, NULL, NULL); // events
```

Spec

Setting kernel arguments:	Section 5.5.2
Executing Kernels:	Section 6.1
Reading, writing, and copying buffer objects:	Section 5.2.2



OpenCL Vector Addition Kernel

```
__kernel void vec_add (__global const float *a,  
                        __global const float *b,  
                        __global float *c)  
{  
    int gid = get_global_id(0);  
    c[gid] = a[gid] + b[gid];  
}
```

Spec

__kernel:	Section 6.7.1
__global:	Section 6.5.1
get_global_id():	Section 6.11.1
Data types:	Section 6.1

OpenCL for Nvidia



- OpenCL is fabulous for developers, the industry consumers and is being driven in large part by **NVIDIA!**
- The addition of OpenCL and DX11 compute to C for CUDA offers a fantastic array of GPU Computing choices for developers
- C for CUDA:
 - Is the only runtime C environment for GPUs today
 - Has 25,000+ users and 100+ applications
 - Will evolve and co-exist with OpenCL and DX11 compute

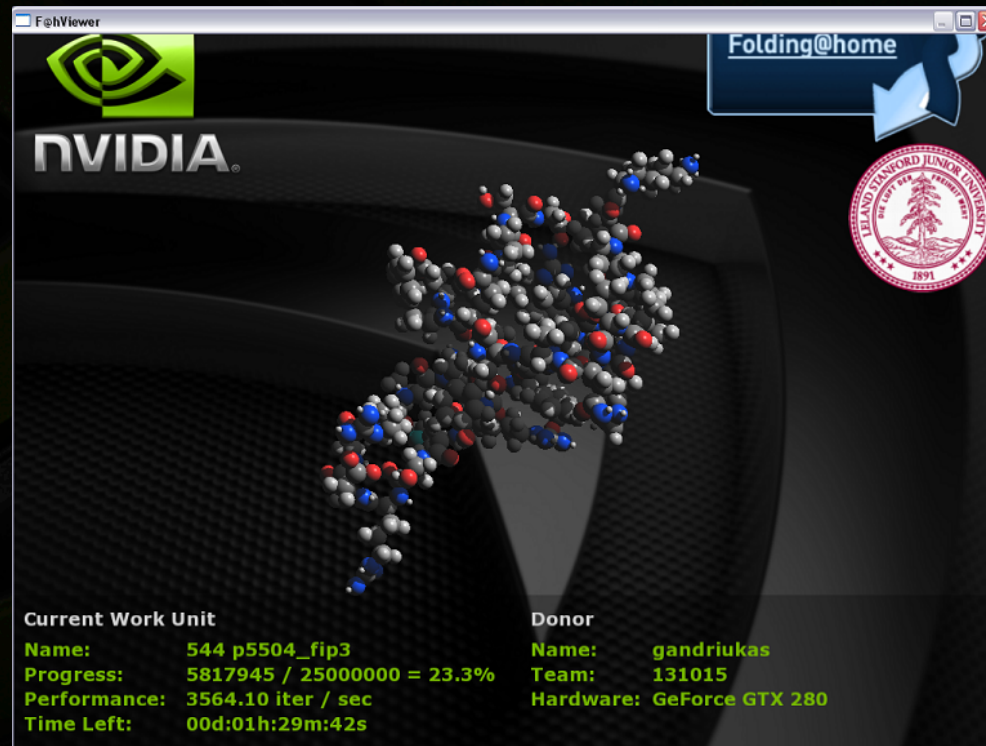
Example Applications



Folding at Home



- **NVIDIA team total production:**
 - top 10% in 2 weeks with 10 GPUs
 - top 0.1% in less than a month with additional GPUs

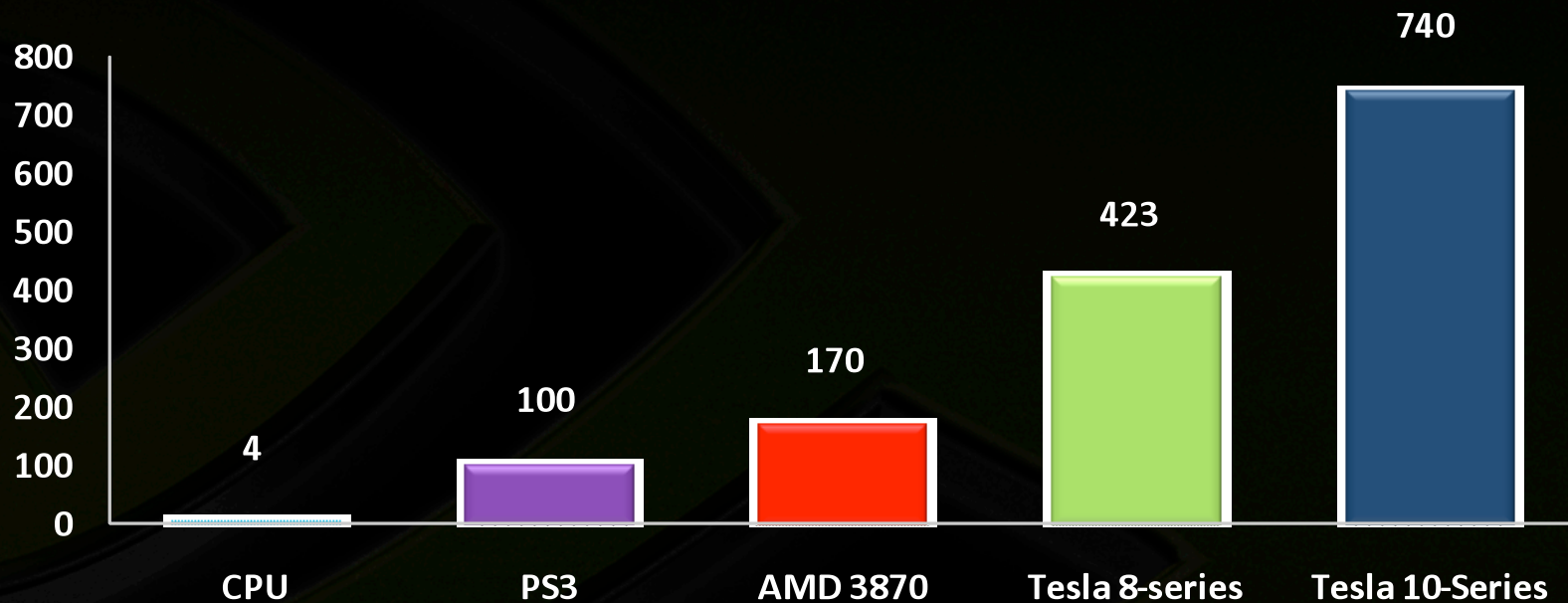
A screenshot of the Folding@home software interface. The window title is "F@hViewer". In the top left corner is the NVIDIA logo. In the top right corner, there is a "Folding@home" logo with a blue double-headed arrow. Below the arrow is a circular seal of Stanford University. The center of the window displays a 3D molecular model of a protein, represented by a cluster of grey, red, and blue spheres. At the bottom, there are two columns of text: "Current Work Unit" and "Donor".

Current Work Unit		Donor	
Name:	544 p5504_fip3	Name:	gandriukas
Progress:	5817945 / 25000000 = 23.3%	Team:	131015
Performance:	3564.10 iter / sec	Hardware:	GeForce GTX 280
Time Left:	00d:01h:29m:42s		

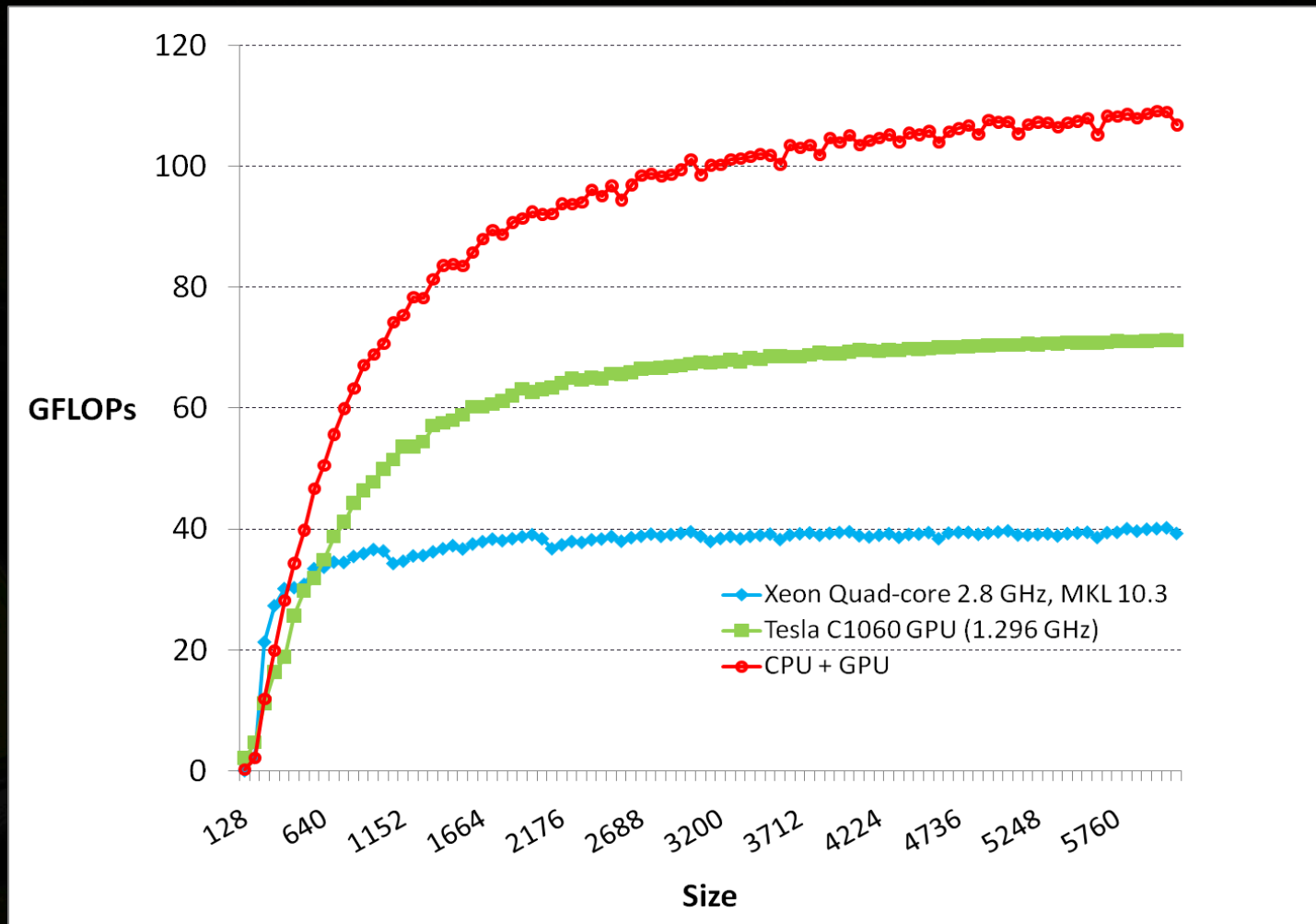
Folding at Home Performance



nano seconds of
simulation per day



DGEMM Performance

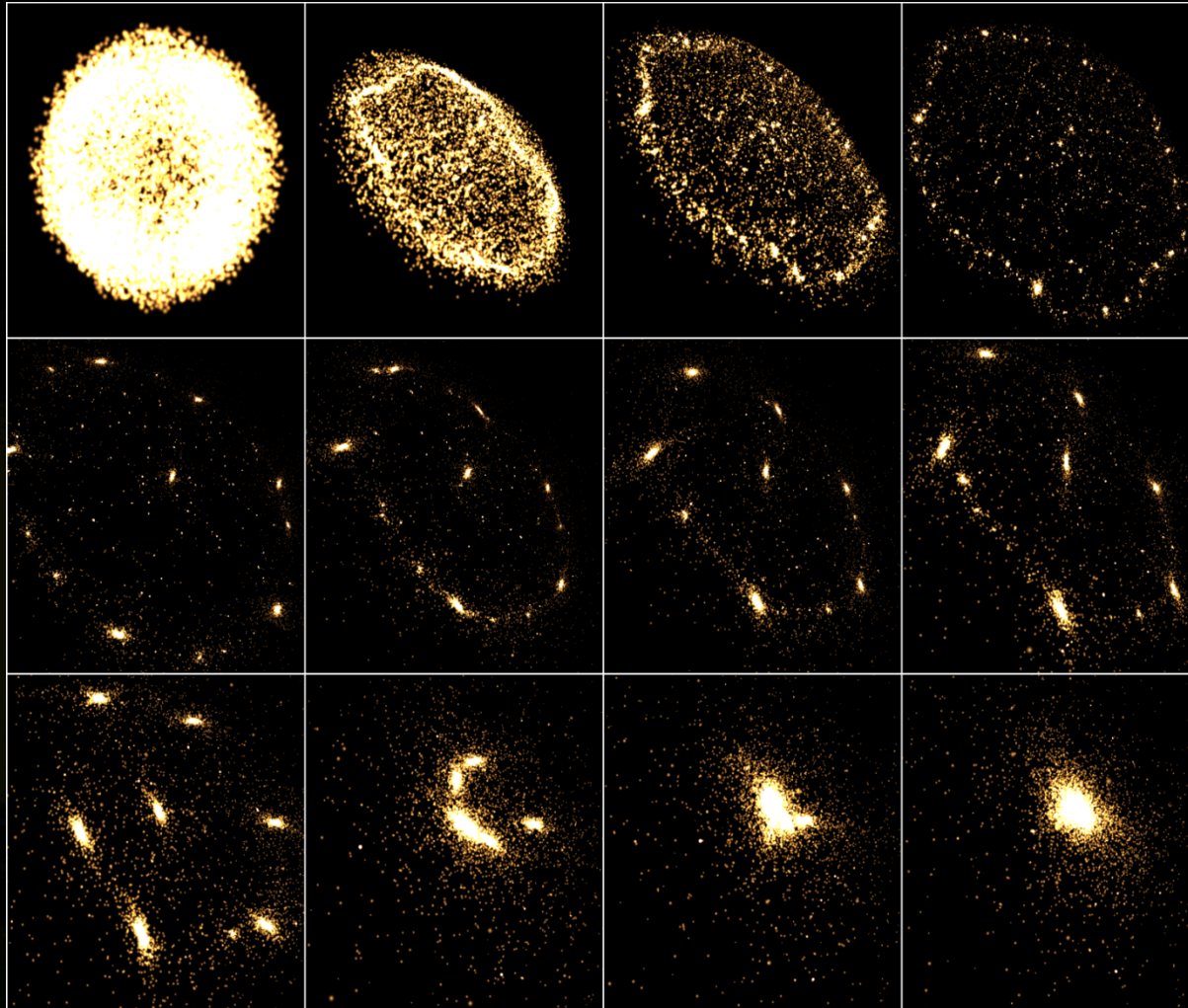


GPU performance includes data copies over PCIe gen-2

CAD Design For Apparel Cloth Physics



CUDA N-Body Simulation



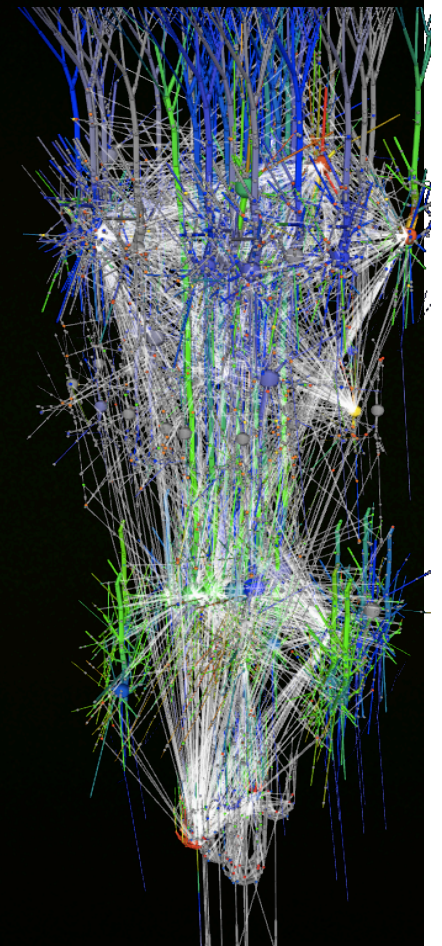
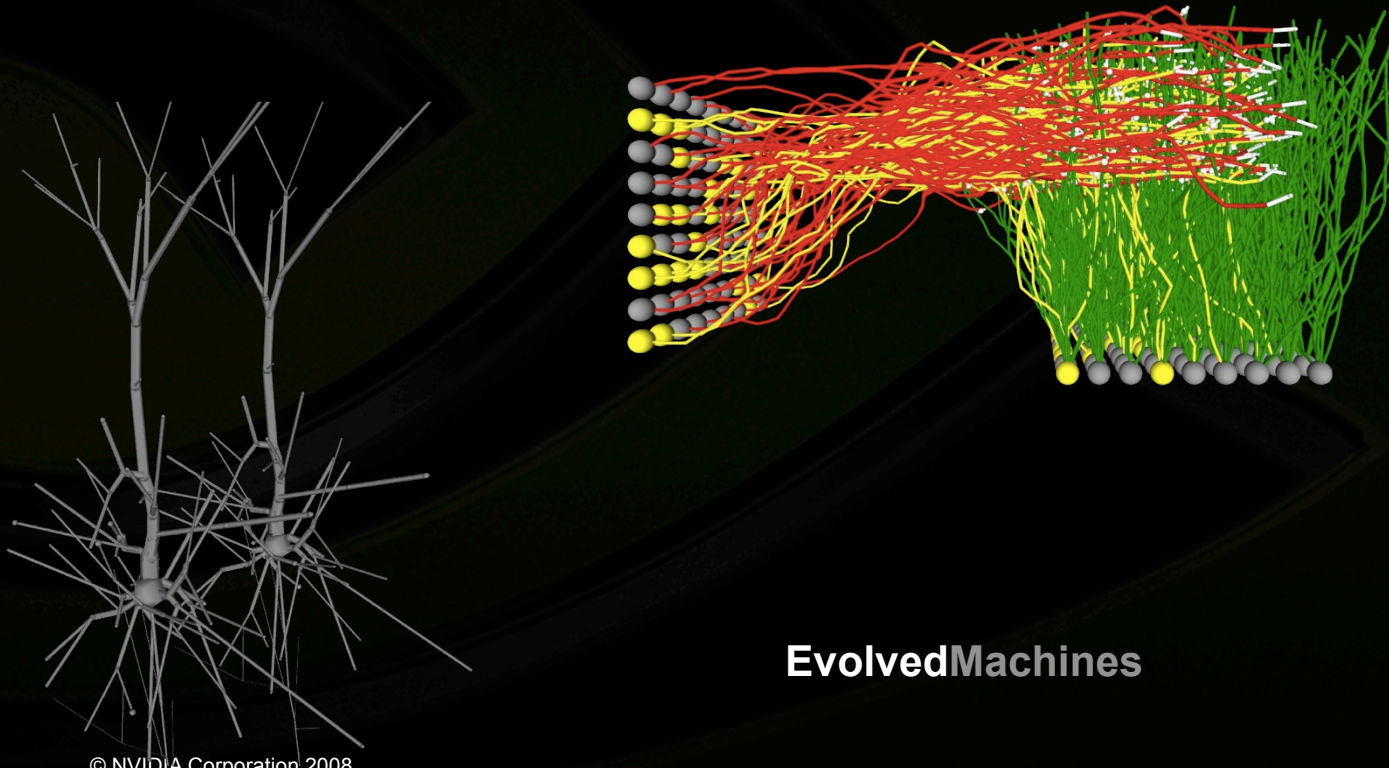
23B interactions/s
30K bodies

GeForce GTX 280:
470 GFlops/s

Evolved**Machines**



- 130X Speed up
- Simulate the brain circuit
- Sensory computing: vision, olfactory



EvolvedMachines

Real-time Ray Tracing

- Real system
- NVSG-driven animation and interaction
- Programmable shading
- Modeled in Maya, imported through COLLADA
- Fully ray traced

2 million polygons

Bump-mapping

Movable light source

5 bounce reflection/refraction

Adaptive antialiasing



Thank You!



- nvidia.com/cuda

- nvidia.com/opengl